Helper Macros for Handler Config

Purpose

To reduce repetitive boilerplate when defining packet handlers, the module also provides a set of helper macros in `packet_dispatcher_macros.h`.

These macros generate:

- a statically allocated queue buffer
- a fully initialized `packet_handler_config_t`

> They are especially useful because they automatically derive the correct queue item size from the selected `PBEnvelope` payload member, which helps avoid one of the easiest mistakes in this module: mismatching `item_size` with the actual decoded protobuf payload type.

Why these macros are useful

Without these macros, every handler config has to manually specify:

- queue storage buffer
- queue length
- item size
- task name
- default priority
- default stack depth
- queue initialization fields

That is tedious and error-prone.

I) They derive `item_size` automatically

Each macro uses: `sizeof(((PBEnvelope*)0)->payload.payload_member)` to compute the exact size of the selected envelope payload member at compile time. This removes the need to manually write `.item_size = sizeof(MyPayloadType)` and reduces the chance of queue item size mismatches.

II) They allocate queue storage automatically

Each macro also declares:

```
static uint8_t name##_queue_buffer[...];
```

with the correct total size based on:

- payload member size
- selected queue length

So the queue backing storage is generated alongside the config object.

Important consequence of these macros

> These macros define **static objects**.

That means each use creates:

- a static queue buffer
- a static `packet_handler_config_t`

> This is generally what you want for a dispatcher configuration that should live for the full lifetime of the system.

It also means:

- they should normally be used at file scope
- using the same `name` twice in one translation unit will cause symbol redefinition
- they are not runtime factory macros, they are compile-time object definition helpers

---

Shared Functionality

For **all** of these macros, the generated config uses:

```
#define PACKET_HANDLER_CONFIG_STATIC(name, packet_tag, payload_member_size, handler_fn)


.handler = (handler_fn)
.task_name = #name
.packet_type = (packet_tag)
```

```
.item_size = payload_member_size
.queue_buffer = name##_queue_buffer
.queue_struct = {0}
.queue = NULL
```

This is helpful for two reasons:

- `task_name` is automatic
  The task name becomes **the same as the symbol (handler config itself) name**, which keeps config definitions compact and readable.
- Queue internals are initialized consistently
  The queue control structure is zero-initialized, and the runtime queue handle starts as `NULL`, matching the expectations of the dispatcher startup code.

IMPORTANT NOTE on `payload_member`

> The `payload_member` argument is not the packet type name. It is the **member name inside `PBEnvelope.payload`!**

This matters because the macros compute size using direct member access syntax: `sizeof(((PBEnvelope*)0)->payload.payload_member)`. **So, if the wrong member name is used, compilation will fail, which is actually helpful for once.**

The member names are defined in `envelope.pb.h` .
For example, currently `envelope.pb.h` contains the following:

```
typedef struct _PBEnvelope {
    pb_size_t which_payload;

    union _PBEnvelope_payload {
            /* Sensorboard messages */
            SensorBoardPHInfo ph_info;

            /* Armboard messages */
            ArmBoardControlSignals arm_ctrl;
            ArmBoardDiagnostics arm_diag;

            //etc etc...
    }
}
```

> So, the macro must be called with the member name **matching the rest of the config**, such as `ph_info` or `arm_ctrl` and **NOT** the protobuf struct type name!

Available macros

1) Default configuration macros

The header defines these default values:

```
#define PACKET_HANDLER_DEFAULT_PRIORITY (tskIDLE_PRIORITY + 2U)
#define PACKET_HANDLER_DEFAULT_QUEUE_LENGTH (5U)
#define PACKET_HANDLER_DEFAULT_STACK_DEPTH (0U)
```

- `PACKET_HANDLER_DEFAULT_PRIORITY`
  Default FreeRTOS task priority assigned to handler tasks created with the simpler macros.
- `PACKET_HANDLER_DEFAULT_QUEUE_LENGTH`
  Default number of queued packets per handler.
- `PACKET_HANDLER_DEFAULT_STACK_DEPTH`
  Default stack depth field stored in the config.
  A value of `0U` is intentional here. In the dispatcher implementation, a task stack depth of zero is treated as “use the dispatcher default,” which becomes: `PACKET_HANDLER_TASK_STACK_DEPTH_DEFAULT`. So this macro does **not** mean “zero stack.” It means “defer to the runtime default chosen by the dispatcher.”

---

2) Basic config: `PACKET_HANDLER_CONFIG_STATIC`

```
#define PACKET_HANDLER_CONFIG_STATIC(name, packet_tag, payload_member_size, handler_fn)
```

This is the simplest form. Creates a handler config using:

- **default** priority
- **default** queue length
- **default** stack depth behaviour

Parameters

- `name`
  User defined name, go crazy.

- `packet_tag`
  This is the Nanopb generated tag for the packet type. They follow the pattern `PBEnvelope_[payload_member]_tag`. So for example: PBEnvelope_arm_ctrl_tag
- `payload_member`
  See important note on payload_member. **Needs to match the packet_tag and the buffer type the callback is specified for!**
- `handler_fn`
  Callback function. Type signature packet_handler_t.

Example

```
/* Config for: ArmBoardMovementFeedback */

//Define the callback function with the specified signature
static result_t Callback_ArmBoardMovementFeedback(void *buffer) {
  if (buffer == NULL) {
    return RESULT_ERR_INVALID_ARG;
  }

  //Retreive the packet
  ArmBoardMovementFeedback* pckt = (ArmBoardMovementFeedback *)buffer;
  //Get all fields
  pckt->arm_error;

  /*
  Go wild...
  */
  return RESULT_OK;
}

PACKET_HANDLER_CONFIG_STATIC(
  Handler_ArmBoardMovementFeedback,   // NOTE: This name is USER DEFINED, let your imagination
run
  PBEnvelope_arm_feedback_tag,        //  Make sure these...
  arm_feedback,                       //                   ... MATCH!
  Callback_ArmBoardMovementFeedback); // Callback as above
```

3) Full config: `PACKET_HANDLER_CONFIG_STATIC_EX`

```
#define PACKET_HANDLER_CONFIG_STATIC_EX(name, packet_tag, payload_member, handler_fn,
                                        priority_, stack_depth_, queue_length_)
```

Full explicit version. Lets you set:

- name, packet_tag, payload_member, handler_fn as above
- **custom** priority
- **custom** stack depth
- **custom** queue length

Best used when

- the handler needs a non-default stack size
- you want fully explicit resource configuration

Example

```
PACKET_HANDLER_CONFIG_STATIC_EX(vision_handler_cfg,
                                PBEnvelope_detected_object_tag,
                                detected_object,
                                handle_detected_object,
                                tskIDLE_PRIORITY + 3U,
                                768U,
                                16U);
```

---

these r not in the code lol

begin here

II) `PACKET_HANDLER_CONFIG_STATIC_QUEUE`

```
#define PACKET_HANDLER_CONFIG_STATIC_QUEUE(name, packet_tag, payload_member_size,
handler_fn, queue_length_)
```

Same as the basic macro, but lets you override queue length.

Best used when

- handler needs a longer or shorter queue
- default priority is still fine

Example

```
PACKET_HANDLER_CONFIG_STATIC_QUEUE(sensor_handler_cfg,
                                   PBEnvelope_sensor_diag_tag,
                                   sensor_diag,
                                   handle_sensor_diag,
                                   12);
```

III) `PACKET_HANDLER_CONFIG_STATIC_PRIO`

```
#define PACKET_HANDLER_CONFIG_STATIC_PRIO(name, packet_tag, payload_member, handler_fn,
priority_)
```

Same as the basic macro, but lets you override task priority.

Best used when

- one handler must run at a different RTOS priority
- default queue length is still fine

Example

```
PACKET_HANDLER_CONFIG_STATIC_PRIO(emergency_handler_cfg,
                                  PBEnvelope_arm_obstructions_tag,
                                  arm_obstructions,
                                  handle_arm_obstructions,
                                  tskIDLE_PRIORITY + 4U);
```

IV) `PACKET_HANDLER_CONFIG_STATIC_PRIO_QUEUE`

```
#define PACKET_HANDLER_CONFIG_STATIC_PRIO_QUEUE(    name, packet_tag, payload_member,
handler_fn, queue_length_, priority_)
```

Lets you override both:

- queue length
- task priority

Best used when

- a handler has non-default scheduling needs
- and also non-default backlog requirements

Example

```
PACKET_HANDLER_CONFIG_STATIC_PRIO_QUEUE(nav_handler_cfg,
                                        PBEnvelope_ph_info_tag,
                                        ph_info,
                                        handle_ph_info,
                                        10,
                                        tskIDLE_PRIORITY + 3U);
```

---

end here

Revision #15
Created 2026-05-18 14:04:23 UTC by Lisa te Braak
Updated 2026-05-24 20:12:23 UTC by Lisa te Braak